Modelling Coordination in Organizations

Kevin Crowston
Center for Coordination Science
MIT Sloan School of Management

December 1990

Sloan WP 3228-90-MSA CCS WP 115

Modelling Coordination in Organizations

Kevin Crowston
Center for Coordination Science
MIT Sloan School of Management

December 1990

Sloan WP 3228-90-MSA CCS WP 115

To appear in M. Masuch and G. Massimo (Eds.), *Artificial Intelligence in Organization and Management Theory*. Amsterdam: Elsevier, 1991.

MODELLING COORDINATION IN ORGANIZATIONS

Kevin Crowston[1]
Center for Coordination Science
Sloan School of Management
Massachussetts Institute of Technology

Introduction

What good are organizations? Obviously, organizations exist for many reasons. One of the most important is to channel the efforts of the organization's members in ways that allow the organization to accomplish things that no individual working alone could do. For example, a computer operating system is such a complex product that no one individual can be said to know how to design or build one, yet several organizations routinely do exactly that. This ability has a cost; much of the work done by members of a software company has little to do with actually writing software. Instead, these workers spend their time *coordinating* their actions and the actions of others. As yet, however, we have only a vague understanding of what coordination work is or how it is useful.

The end goal of my research is to provide a more principled definition of coordination and coordination work. To do so, however, requires the development of better analysis techniques. In this paper I will present such a technique and briefly discuss its implications for study of organizations and coordination.

Coordination seems to be primarily an information processing task. One method other information-processing-based disciplines have used to gain insight into complex behaviours is to imagine how a computer could be programmed to reproduce them. In cognitive psychology, for example, computer models of learning or memory have been used to make theories about human information processing concrete and to generate further empirically testable hypotheses.

---

1 Author's present address: E53-322, MIT, Cambridge, MA 02139 USA
Electronic mail (internet): kevin@xv.mit.edu
Telephone: +1 (617) 253-2781
Fax: +1 (617) 253-7579

Cyert and March (1963) took a similar approach to the study of organizations. In their analysis of the process firms used to make pricing decisions, the "process is specified by drawing a flow diagram and executing a computer program that simulates the process in some detail" (p. 2).

Computer models of organizations can provide many benefits for the study of coordination in organizations. First, artificial intelligence research and in particular the developing field of distributed artificial intelligence (DAI, *e.g.*, Huhns, 1987; Bond and Gasser, 1988; Gasser and Huhns, 1989; Huhns and Gasser, 1989) can contribute interesting formalisms for understanding and representing the actions of human organizations. Second, computer systems are a much more tractable method for investigating organizations. For example, it is possible to perform true experiments comparing systems of coordination using computer models (*e.g.*, Durfee, 1988). In a sense, DAI is (or can be) the experimental branch of organizational science.

In this paper, I concentrate primarily on the first of these potential contributions. Using ideas from DAI, I have developed a technique for modelling the coordination processes of a group performing a complex task. I focus in particular on the additional knowledge an individual working in a group needs to know to be an effective member of the group beyond simply knowing how to do his or her job.

Example: Computer software company

The technique I will be describing was developed in the course of a field study of the engineering change processes in three large manufacturing companies. To clarify this context, I will briefly describe one of these companies, which will provide examples for the rest of the paper.

Description of site

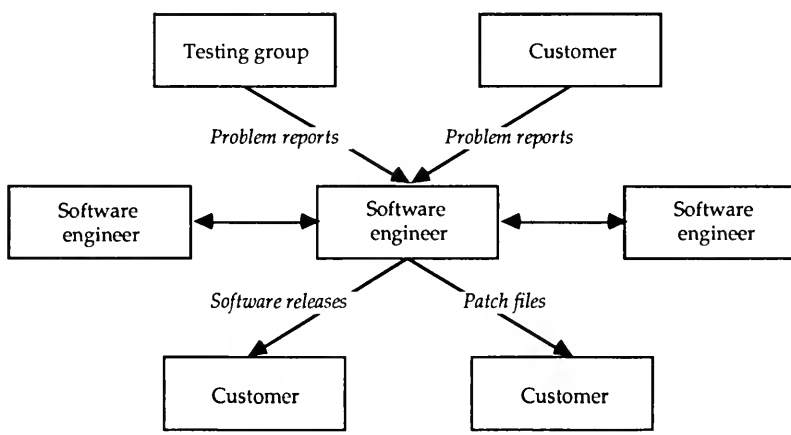
The company I studied is a manufacturer of computer hardware and system software; the particular group I studied was responsible for the development of the kernel of the operating system, a total of about one million lines of code in a high-level language. The group had a total of about 200-300 programmers, divided between 8 development groups and various support groups. The operating system was divided into numerous modules; each software engineer was responsible for a few of these modules.

Overview of change process

Lientz and Swanson (1980) describe three kinds of software changes: corrective, perfective and adaptive. Corrective changes are those made to fix problems with the software, defined by this company as disagreements between the documented and actual behaviour of the software. (Note that these problems may be fixed by changing either the documentation or the code.) Perfective changes are those made to improve the software (for example, by improving performance) without adding new functionality. The group also made adaptive changes, adding new functionality for future releases, but I did not study these changes. However, the organization was clearly shaped in large part by the need to perform these changes.

The basic flow of a change is shown in Figure 1. Changes are made in response to problem reports, which come from a testing group and from end users of the operating system. These reports are filtered and genuine and novel problems are routed to the software engineer responsible for the apparently affected module. This engineer, in consultation with other engineers, develops a fix for the problem. The fix may require changes to other modules; those changes are made by the engineers responsible for the other modules. Customers get problem fixes either periodically as part of a new release of the software or, for more urgent problems, as a separate patch file that can be loaded on top of the current release.

Figure 1. *Overview of change process for example case.*



Underlying theoretical assumptions

As is apparent even from this brief description of the example research site, the change process involves many actors and subtasks. Models are useful because they can be used to abstract from and simplify complex systems such as this one. As Yourdon notes, "we can construct models in such a way as to highlight, or emphasize, certain critical features of a system, while simultaneously de-emphasizing other aspects of the system" (1989, p.65). The key issue in modelling, then, is the choice of which elements of the observed phenomena to include and which to omit. Which variables should appear in the model? How should they appear? What are the appropriate values? Different modellers have chosen different answers to these questions. To suggest which features are important to include and which are unimportant details, it is necessary to have a strong theoretical view of the organization.

I adopted the information processing (IP) view of organizations (*e.g.*, March and Simon, 1958; Galbraith, 1977; Tushman and Nadler, 1978) because of its focus on how organizations process information. Tushman and Nadler (1978, p. 292) outlined three basic assumptions of IP theories: (1) organizations must deal with work-related uncertainty; (2) organizations can fruitfully be seen as information processing systems; and (3) organizations are composed of individual actors. In this view, organizational structure is the pattern and content of the information flowing between the actors and the way they process this information. I adopted a quite reductionistic version of this view, focusing exclusively on modelling the information-processing behaviour of the individual actors that comprise the organization and the communications between them (Prietula, Beauclair, and Lerch, 1990).

In order to provide a focus for and boundaries around the models, I assumed that the organization had a clear goal it was trying to achieve. In the example case, the goal seems to be to implement engineering changes to fix existing problems without introducing new problems. I am not claiming that the actors perform only this one particular task; clearly actors have many goals, both organizational and individual. However, I am analyzing the organization's performance with respect to only certain of these goals. These goals are attributed to the organization by the analyst for the purposes of a particular study; other analysts or the actors themselves may not necessarily agree that they are the correct (or most interesting) goals. For example, an analyst might choose to view the goal of a market as the efficient allocation of scarce resources to different actors. None of the actors in the market necessarily have "efficient

allocation of resources" as a goal, but it still makes sense to ask how efficient a particular market structure is for achieving that goal.

In the rest of this paper I will describe the steps I went through to model the organizations I studied. The description will be illustrated with examples from the case site described above. I will conclude by discussing possible uses for such models.

Data-flow models

Because coordination seems to be primarily an information-processing task, I wanted first to clearly identify the information processing done by the members of the organization. As in an earlier study (Crowston, Malone, and Lin, 1987), I approached this problem by developing what I call data-flow models. These models are similar to data-flow diagrams (see, for example, Yourdon, 1989) or the structured analysis and design technique (see Marca and McGowan, 1988).

Data-flow models include two major elements: actors and messages. Actors send messages to other actors. When an actor receives a message, it takes some action, which may include sending additional messages to other actors. Each kind of actor understands and reacts to a different set of messages. I use the term "message" here in an abstract sense that includes any kind of communication, verbal as well as paper or electronic. The resulting model is similar to a program written in an object-oriented language (*e.g.*, Stefik and Bobrow, 1986). The object-oriented metaphor suggests creating a hierarchy of actor types as a way to simplify the description of different actors and to highlight their similarities; this method was used in (Crowston et al., 1987).

For example, there might be a number of individuals designing modules of an operating system, all working in roughly the same way and using the same kinds of information; each would be modelled as an example of a "software engineering actor". A testing actor works differently and would be analyzed separately. Engineering actors receive notifications of changes in other modules from other engineering actors or of problems in their modules from testing actors. When an engineering actor receives a change notice, for example, it first determines if the notice affects its modules, based on its knowledge of the connections between modules; if it does, the actor then makes any necessary changes and send the revised module and its own change notices to other actors.

The product of this analysis is a specification of the types of messages and the behaviour of the actors, in principle in precise enough detail to construct a computational model of the organization. In presenting the model, I give only a description of the different types of actors (the classes) and the actions they take for each kind of message they understand; I do not present a full instantiation of the model. A working simulation would have an object of the appropriate class to represent each actor in the modelled organization.

Data collection

Perhaps the most important function of the data-flow models is to provide a focus for data collection. In developing the models presented here, I used an iterative approach, sometimes called negative case study method (Kidder, 1981), switching between data collection and model development. The initial round of data collection served as the basis for an initial model. Constructing this model would reveal omissions in the data, for example, places where it was not clear how an actor would react to some message or from whom a particular piece of information came. These omissions or ambiguities served as the basis for further data collection.

To collect data for these models we can make use of techniques developed in fields ranging from ethnography to expert system knowledge acquisition (*e.g.*, Ericsson and Simon, 1984). In my case, most of the data collection was done in one hour or longer semi-structured interviews with various members of the organization. As March and Simon (1958) pointed out, "most programs are stored in the minds of the employees who carry them out, or in the minds of their superiors, subordinates or associates. For many purposes, the simplest and most accurate way to discover what a person does is to ask him" (p. 142). Data was collected by asking subjects questions such as: (1) what kinds of information they receive; (2) from whom they receive it; (3) how they receive it (*e.g.*, from telephone calls, memos or computer systems); (4) how they process the different kinds of information; and (5) to whom they send messages as a result. I attempted to behaviourally ground these questions by asking interviewees to talk about the events that had recently occurred and using them as a basis for further questions. For example, I asked some individuals to go through their in-boxes and describe the different kinds of documents they found and what they did with each one (*e.g.*, Brobst, Malone, Grant, and Cohen, 1986; Malone, Grant, Turbak, Brobst, and Cohen, 1987a).

Relying on interviews alone can introduce some biases. First, people do not always say what they really think. Some interviews were conducted in the

presence of other employees of the company, so interviewees might have been tempted to say what they thought they should say (the "company line"), what they thought I wanted to hear or what they thought would make them or the company look best. Second, individuals sometimes might really not know the answer. I tried to control for some of these biases by checking reported data with other informants. I also attempted to collect more objective information, such as data about: (1) the types of information stored in computer systems; (2) the use of computer systems; (3) names on memo distribution lists; or (4) examples of forms used.

Another source of data was documents describing standard procedures or individual jobs, such as training manuals. March and Simon (1958) suggest that these documents are created for three different reasons: (1) as instructions for individuals doing the job, (2) as descriptions for new members of the group and (3) to legitimize or formalize the procedure (p. 142). They note that interpretation of these documents depends on the purpose they were intended to serve.

Finally, to get a better sense of the kinds of communication individuals actually used, I observed some individuals during the course of a typical work day. For example, in one site I followed engineers for a day, during which I sat in on scheduled and unscheduled meetings and took notes about the kinds of people the engineer interacted with and the types of information exchanged.

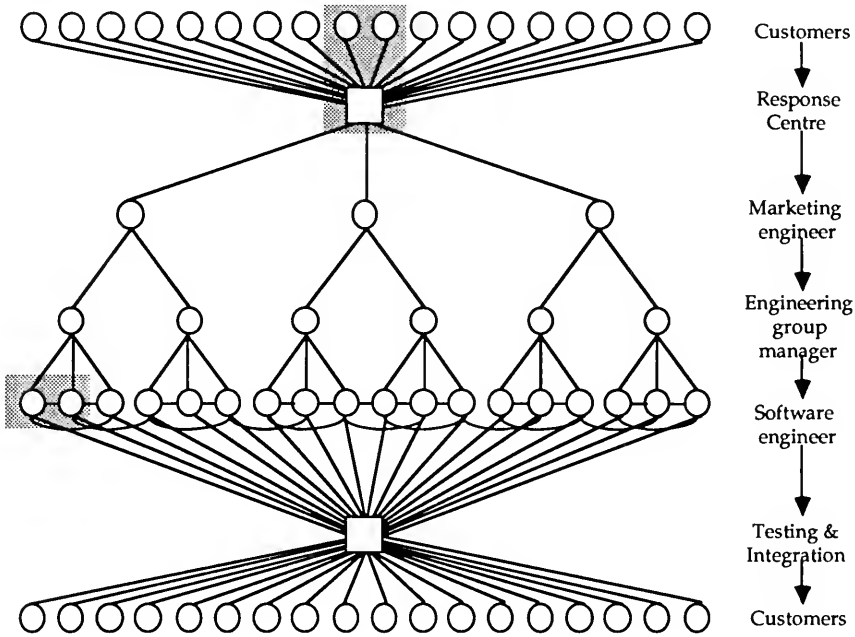
It is interesting to note that the process apparently followed frequently differed from the formal process. For example, at one site, engineers received a listing of all approved changes, but the official list merely confirmed that the changes had been approved. In order to react to a change, the engineer had to be warned of it well in advance of its appearance on the official list. This warning seemed to happen primarily through an informal process. It was this informal process that I attempted to model.

Example

An overview of the information flow model for the software change process is shown in Figure 2. Messages flow generally from top to bottom. Only the primary information flow is shown in the diagram; messages do occasionally skip levels or flow backwards to provide feedback, but these links are not shown.

Six different kinds of actors are shown: customers, the response centre, marketing engineers, engineering group managers, software engineers and

Figure 2. Overview of information-flow for example case.



testing and integration. Note that these "actors" include both individuals, such as customers or software engineers, and groups, such as the response centre or testing and integration. I belive that the ability to mix levels of analysis in this way is an important advantage of this sort of modelling. The response centre (to pick a group) is, of course, composed of many individuals and has its own somewhat complex internal structure and processes. However, for the purposes of analyzing the engineering change process, I decided that the details of this internal structure were less interesting than, for example, the interactions between engineers. I therefore chose to hide the internal details of the response centre in a black box, modelling it as a single actor, while focusing in more detail on the interactions between engineers. This choice would be made differently if the model were used for to investigate other questions. For example, knowing the internal structure of the response centre might be quite important for analyzing cases where customer complaints seemed to get lost or misrouted.

Obviously, the model is too big to discuss in detail in the space available. I will concentrate, therefore, on the two highlighted parts of the process: the

interactions between a customer and the response centre and between software engineers. These parts were chosen because they give a good feel for how the modelling process works and will be reasonably familiar to readers acquainted with the process of software maintenance.

Tables 1 and 2 show the messages understood by the response centre and software engineers. Customers send *Problem Report* messages to the response centre. When the response centre receives the message, it attempts to determine if the problem is due to a customer misunderstanding; if it is, then an explanation can be immediately returned. If the reported problem duplicates a known problem, one that appears in a database of previous calls or reported problems, then the earlier response can be reused. If the problem can not be solved in this way, then the *Problem Report* message is resent to the marketing engineer for the product.

Software engineers understand a wider variety of messages. Initially they receive *Problem Report* messages from the manager of their group (who in turn receives them from the marketing engineer). Engineers first attempt to locate the problem in a particular module. If it appears that the problem is actually in a module other than one for which the engineer is responsible, then the *Problem Report* message is resent to the engineer responsible for the apparently affected module. Otherwise, the engineer puts the problem on a queue of problems to be fixed and picks the most important problem to work on next.

Engineers usually discuss proposed changes with each other before they are implemented. This process is modelled as an exchange of *Proposed Solution* and *Comment* messages. (Note that we are attempting to model the kinds of interactions necessary, not the actual content of any particular exchange.) When an engineer has developed a solution for a problem, he or she determines which modules and therefore which engineers are likely to be affected and send those engineers a *Proposed Solution* message. The engineers then return *Comment*

Table 1. Messages understood and actions taken by Response Centre.

Sender	*Message*	*Recipient*	*Actions taken*
Customer	*Problem report*	Response Centre	if customer misunderstanding, return explanation if duplicate problem, return known solution otherwise, send *Problem report* message to the marketing engineer for this product

messages. If the comments are positive, then the engineer implements the change (that is, actually writes the code and changes the module); otherwise, he or she revises the proposed solution and goes through the comment process again.

Once the change has been implemented, the changed module is tested and submitted to the testing and integration group to be included in the next release of the system. High priority changes (the priority is set by the marketing engineer based on the customer's report) may be issued as a patch file by the patch coordinator. These processes are modelled by having the engineer send a *Solution* message to the appropriate actors.

Implementation of a change may require other engineers to change their modules. For example, new functionality may be required from another module to support the change. This process is modelled by having the first engineer send *Problem Report* messages asking the engineers responsible for the affected

Table 2. Messages understood and actions taken by Software engineers.

Software group manager or other Software engineer	*Problem report*	Software engineer	locate problem if in different module, send *Problem report* message to appropriate engineer prioritize the problem, and work on most important determine the change necessary send *Proposed solution* message to affected engineers and wait for comments.
Software engineer	*Proposed solution*	Software engineer	return appropriate *Comment* message
Software engineer	*Comment*	Software engineer	if negative, then revise and resend *Proposed solution* message otherwise, implement proposed solution send *Problem report* message to engineers requesting necessary changes to other modules send *Solution* message to Integration if problem is high priority, send *Solution* message to Patch coordinator

modules for the appropriate changes. Those engineers independently submit their changes directly to the testing and integration group or to the patch coordinator with an indication that they are part of the initial change.

Intentional models

While the information-flow models usefully abstract the communication between actors and the way they process this information, they do not adequately explain why the actors communicate in the ways they do and not in other plausible ways. As Newell and Simon (1972) noted in their analysis of individual problem solving, the flowcharts just appear, with no real explanation as to their origins. Following their example, the next step of my analysis was to develop a problem-solving model for each actor that generates the observed communications. I call these models *intentional models*, since they capture the intentions behind the actors' actions.

To do this modelling, I again drew on ideas from DAI. I modelled each actor as an independent goal-directed problem solver. Each actor is assumed to have its own knowledge about the world; actors can communicate but do not directly share memory. Each actor attempts to achieve its goals, given the state of the world as it knows it, by taking actions that affect that state. To know to ask another actor for help, actors must be able to reason about other actors' knowledge and capabilities. I therefore represented each actors' goals, capabilities and knowledge about task domain and its models of other actors. Essentially, I attempted to reverse engineer the knowledge actors use from the messages they send to other people.

I represented each actor's knowledge as a set of well-formed formulas in first-order predicate calculus. Using logic as a basis for a knowledge representation scheme is common in artificial intelligence research and although it is not universally accepted, it is sufficient for my purposes and I will not examine the alternatives here. (For a better defence of the utility of logic see (Hayes, 1977; McDermott, 1978; Moore, 1982)). It should be noted that none of my results depend crucially on the use of logic as a representation.

Representing knowledge about actions

In order to accomplish their goals, actors perform actions. For example, the action Fix-symptoms(s) represents the actions of some actor developing a fix

for the symptoms s. Note that Fix-symptoms(s) represents the whole class of actions of fixing symptoms.

Actions have preconditions which must be satisfied for the actor to be able to perform the action. These actions fall into two classes: knowledge preconditions and physical preconditions (Moore, 1979; Morgenstern, 1987; Morgenstern, 1988).

Knowledge preconditions capture the idea that actors need knowledge in order to perform actions. First, an actor must know how to perform an action, in this case, how to fix symptoms. Some actions are primitive and can be performed by all actors. More complex action may be performed only by actors who know how to decompose the action into primitive actions. Second, an actor must know the parameters of the action, in this case, the symptoms themselves.

Some actions have physical preconditions that must also be satisfied. For example, finding a fix for the symptoms may require the use of a computer terminal, being able to type or program, etc. For my examples, however, physical preconditions are minimal and will not be discussed.

Once performed, actions have effects; actors use their knowledge of these effects to reason about which actions to undertake to achieve their goals. Of course, actions may also have unanticipated effects or the result may be unknown until the action is performed (e.g., when testing for some condition).

A particularly important set of primitive actions are communications actions. For example, one actor may tell a second actor some fact (represented by Inform(speaker, hearer, fact)) or request that the second actor perform some action (Request(speaker, hearer, action)). As a result of these actions, the second hearer knows the fact or has a goal of performing the action.

In the models, I represent what an actor knows about the preconditions and effects of an action. To simplify the specification of actions, I usually do not provide the decomposition for these complex actions; rather, I simply note that a particular actor is capable of performing the action. In principle one could work out in detail the primitive actions and knowledge necessary to perform each of the actions in the model. (In fact, knowledge engineers do exactly this when they develop an expert system.) For the purpose of these models, however, such detail is usually unnecessary. It is important to know, for example, that software engineers can locate problems in particular modules (and that other actors can not); it is not essential to know in detail how they do that. Representing this bit

of task knowledge abstractly greatly simplifies the development and representation of the model.

Actors may also know that another actor can perform an action without knowing how to do it themselves. Customers, for example, know that response centre can solve problems, but they do not know in any detail how this is done. Nevertheless, they can reason about how to take advantage of this ability.

Example

As an example of this approach, the model for the Customer is shown in Table 3. Note that the model includes both the actor's own knowledge, capabilities and goals as well as its beliefs about the knowledge and capabilities of the other actors with which it interacts (in this case, the Response Centre).

The model for the customer is simple, reflecting a deliberate effort on the part of the company to make it easy for customers to report problems. Customers know the symptoms of the problem and how important the problem is to them; they have a goal of knowing a solution to the symptoms; but they do not, themselves, know anything about how to find solutions. (More precisely, the process does not require or make use of any knowledge the customer may have about how to fix the problem. Knowledgeable customers may therefore try to find other ways to report problems and employees of the company may try to

---

Table 3. Intentional model of a Customer (C).

$\exists$ s : Symptoms(s) $\wedge$ Know(C, s) $\wedge$ Know(C, Importance-of-problem(s))
Customers know some symptoms and how important the problem is

Can(C, Talk-to(C, Response-centre))
Customers can talk to the response centre

Symptoms(s) $\wedge$ Know(C, s) $\Rightarrow \exists$ f : Fix(f) $\wedge$Want(C, Know(C, f) $\wedge$ Fixes-symptoms(s, f))
Customers' goal is to know a fix that fixes the symptoms

Know(C, Symptoms(s) $\Rightarrow$ Can(Response-centre, Fix-problem(s)))
Know(C, Does(x, Fix-problem(s)) $\Rightarrow$ Knows(x, f) $\wedge$ Fixes-symptoms(s, f))
Customers know that the response centre can find a fix for the symptoms

---

find ways to exploit the customer's knowledge.)

Customers do know, however, that the response centre can find a solution that fixes the symptoms as long as they know the parameters to the action, namely, the symptoms. Furthermore, they know how to communicate with the response centre. Therefore, the customer provides the response centre with the symptoms and asks it to solve the problem by performing the Fix-problem action. Performing this action results in the response centre knowing the action, so the customer then asks to be told the solution for the problem. This process in shown in Table 4. In the information-flow model, these three communication acts are all included in the *Problem Report* message from the customer to the response centre.

Uses of models

Each intentional model shows what the individuals in the organization need to know to coordinate with the other members of the organization. For example, in the organization discussed, the customer only needs to know the capabilities of the response centre and how to communicate with it, while a typical software engineer needs (among other things) a fairly detailed model of the interdependencies between parts of the operating system and a way to map between the modules of the operating system and the capabilities of other engineers.

Table 4. *Customer communication behaviour.*

Inform(Customer, Response-centre, s)
The customer tells the response centre the symptoms (the parameter to the action).

Request(Customer, Response-centre, Fix-problem(s))
The customer asks the response centre to fix the symptoms.

Request(Customer, Response-centre, Informref(Response-centre, Customer, λxFixes-symptoms(s, x))
The customer asks the response centre to send it the fix (more precisely, to perform the action of informing the customer of some x that satisfies the relationship Fixes-symptoms(s, x)).

Designing computer-support systems

Knowing the coordination knowledge needed by an actor in a particular organization may itself be directly useful. For example, the model may suggest possible information systems to support the members of a group performing the task by suggesting what information should be provided to help them coordinate.

For example, one coordination problem faced by software engineers in the change process is determining which other engineers might be affected by a proposed change. An information system could help by notifying them of interactions with modules they might otherwise overlook or by helping them find the engineer responsible for a particular module. Currently, engineers seem to independently and informally keep track of which other engineers use the interfaces they maintain. This kind of information is easy to maintain manually and in a decentralized fashion when the development group is small and physically close together but the task becomes more difficult as the group grows and as other, more remote groups are involved.

One way to implement such a system would be a database of interfaces and users; an engineer planning to modify an interface could use the database to determine who should be notified. However, such a database might quickly become out of date. If so, it would be no worse than the current system but would offer few advantages and would therefore probably not be used. (In fact, one of our interviewees had developed a database of interfaces, but had decided not to use it until a better system could be devised to keep it current.) A more useful system would include better methods and motivations for users to register their use of interfaces. For example, having a tool that noticed when a new interface was used might make it easier for an programmer to know to register as a user of the interface. Alternately, a system could compute the interdependencies directly from the code of the operating system, thus guaranteeing a complete list.

New organizational designs

A model could be used as a basis for various kinds of organizational redesigns. Once we understand how an organization is currently coordinated and the constraints that led to a particular organizational structure, we may be able to design organizations to perform the task that coordinate in entirely new ways. As the increased use of information technology makes coordination cheaper, these new organizational forms may become more desirable.

For example, in the organization studied, programmers were specialists; to assign a problem report to an actor required determining what part of the system was involved and then routing the problem report to the appropriate actor. Other divisions of the same company used generalists actors; an incoming bug report was simply assigned to the next available actor to be fixed. (These two bases for organizing are sometimes called module ownership and change ownership (Embry and Keenan, 1983).) Malone, Benjamin & Yates (1987b) discuss the increased use of market-like transactions as a possible consequence of cheaper coordination. One can imagine using a market-like system instead of specialists to assign change requests.

In such a system, whenever a change request was received, it would be broadcast to all maintainers. Maintainers would bid on the changes they wanted to perform and the change would be assigned to the lowest bidder (*e.g.*, least time to perform the change or earliest finish date or even lowest cost). If the change required some specialized knowledge, then maintainers with that knowledge could bid lower and thus be assigned the task. This system would ensure that each change is worked on by the person best suited for it at the time, thus reducing the total cost of making changes. Large changes that required multiple workers could also be handled this way: a single maintainer who was good at project management could bid on the change and then decompose it, subcontract the subtasks and integrate the results.

The coordination cost of this scheme is increased by the need to broadcast bug reports and manage the bidding process, but a computer conferencing system, for example, could make these processes quite inexpensive. A more important problem is the duplication of effort necessary for each maintainer to assesses each change request and determine how much to bid. In addition, there may be agency costs to consider (Ross, 1973); for example, if actors are paid regardless of how many bugs they fix, then they may be motivated to bid high to avoid work; if it is difficult to measure the performance of the actors (*i.e.*, how well the problem has been fixed) then a maintainer could bid low to win work but not actually fix the problems.

Coordination and organization theory

More importantly, I plan to use the models as basis for building theory about organizations, generalizing from the few cases I have studied to a more generic description of coordination problem and methods. For example, task assignment seems to arise as a component of many group tasks, and can be

performed in at least three ways, as discussed above. A typology of situations that require coordination and the set of methods that can be used in these situations would provide a much more succinct language for describing organizational processes as well as a set of pieces from which to design new organizations.

A promising basis for such a typology is the use of boundary objects (Star, 1989). Malone and Crowston (1990) suggest that the need for coordination arises from constraints imposed on the performance of tasks by the interdependencies between the tasks. These interdependencies, in turn, can be analyzed as arising from the tasks' mutual use of common objects. Some of these mutual uses are inherent in the definition of the tasks; for example, the output of one task (writing the code for a particular module) may be the input to another (integrating the system), resulting in what we call a *prerequisite* constraint (similar to what Thompson (1967) calls a sequential interdependence). Other interdependencies arise from the way tasks are assigned to resources, including actors; for example, two tasks may both require a particular tool, resulting in a *shared resource* constraint if there is only one such tool.

Given these constraints, coordination is the work necessary to overcome them. For example, to overcome a prerequisite constraint at least one of the actors must be aware of the constraint and know how to communicate with the other party. Furthermore, the actors must ensure that the tasks are done in the correct order. For a shared resource constraint, the actors must be able to schedule their use of the resource to avoid conflicts.

By identifying the shared objects and the way they are used in different tasks, we can understand the communication in terms of what the actors do to the objects, *e.g.*, negotiating what the details of the object should be or passing an object from one actor to another. For each such situation, there may be only a few patterns of messages that are exchanged. For these sorts of investigations, comparisons between different organizations will be particularly illuminating.

For example, a customer call object is created by a customer using the system and consumed by the response centre in handling customer complaints. This pattern, where one task creates an object that is consumed by another, is handled in this case by having the customer simply send the object (as a *Problem report* message) to the response centre. The exchange of *Proposed Solution* and *Comment* messages can be interpreted as a negotiation between software engineers about the details of another shared object, a Proposed Change, that is used as an input by the tasks of changing the affected modules.

Alternative coordination strategies

Knowing the constraints may suggest alternative ways to manage them. For example, there seems to be at least three basic ways to handle the interdependence between the customer and the response centre. At a minimum, the user must create the customer call and send it to the response centre to answer. Second, the user and the response centre can negotiate the details of the call, for example, by iterating the process (*i.e.*, the customer files a complaint, the response centre asks for more details, the customer supplies them, *etc.*) or in a continuous dialogue. Third, some of the knowledge about the constraints of either task can be moved from one actor to another.

This final case has three interesting subcases. First, some of the customer's knowledge can be transferred to the response centre, for example, by having the response centre recreate the situation on their own computers. Second, some of the response centre's knowledge can be made available to the customer. For example, at the site studied, a computer system had recently been developed that allowed customers to do their own searches through the database of known problems. Finally, a third party may have some of both actors' knowledge and be able to mediate between them. In some cases, for example, the customer engineer responsible for the site may investigate the problem and file a change request.

This analysis seems to be easily transferable to other settings. For example, between the design and manufacture of a part, the common object is the design of the part. Again, the designer must at least provide the manufacturer with the design. Alternately, the designer and the manufacturer can negotiate the details of the design, in several possible ways. Finally, some of either actor's knowledge can be given to another actor.

Again, there are three subcases. First, some of the manufacturer's knowledge (knowledge about the manufacturing constraints, not about how to do the manufacturing) can be made available to the designer, for example, by training the designer in methodologies such as design for manufacturing or by embodying the knowledge in an intelligent CAD system. Second, some of the designer's knowledge can be transferred to the manufacturer. For example, if the design captures the designer's intent as well as the details of the part, the manufacturing engineer might be able to change some details of the design to make the parts easier to build while preserving the intent. Finally, a third party,

such as a common superior, may have some of both engineers' knowledge and be able to mediate between them.

Conclusion

In this paper, I have described a method for modelling coordination based on coordination knowledge and given examples of its use in a field study. I believe that understanding the coordination needs of different tasks may help in design of coordinated systems and organizations.

In the future, I plan to use such models as a basis for a computer simulation of the organizations. Using such models, I hope to be able to explore more systematically the implications of different distributions of knowledge and capabilities among actors. The development of these computer simulations may lead to the development of a body of coordination methods, comparable to the weak methods of individual problem solving and useful for any group task.

References

Bond, Alan H., and Les Gasser (eds.)
1988 Readings in Distributed Artificial Intelligence. San Mateo, CA: Morgan Kaufman.

Brobst, Steven. A., Thomas W. Malone, Kenneth R. Grant, and Michael D. Cohen
1986 "Toward intelligent message routing systems." In Computer message systems-85: Proceedings of the Second International Conference on Computer Message Systems: 351-359. Amsterdam: North-Holland.

Crowston, Kevin, Thomas W. Malone, and Felix Lin
1987 "Cognitive science and organizational design: A case study of computer conferencing." Human Computer Interaction, 3: 59-85.

Cyert, Richard. M., and James G. March
1963 A Behavioral Theory of the Firm. Englewood Cliffs, NJ: Prentice-Hall.

Durfee, Edmund H.
1988 Coordination of Distributed Problem Solvers. Boston, MA: Kluwer Academic.

Embry, J. D., and J. Keenan
1983 "Organizational approaches used to improve the quality of a complex software product." In R. S. Arnold (ed.), Software Maintenance Workshop: 131–133. Monterey, CA: Naval Postgraduate School.

Ericsson, K. Anders, and Herbert A. Simon
1984 Protocol Analysis: Verbal Reports as Data. Cambridge, MA: MIT Press.

Galbraith, Jay R.
1977 Organization Design. Reading, MA: Addison-Wesley.

Gasser, Les, and Michael N. Huhns (eds.)
1989 Distributed Artificial Intelligence. San Mateo, CA: Morgan Kaufmann.

Hayes, Patrick J.
1977 "In defence of logic." In Proceedings of the Fifth International Joint Conference on AI (ICJAI-77): 559–565. Cambridge, MA: Available from the Department of Computer Science, CMU.

Huhns, Michael (ed.)
1987 Distributed Artificial Intelligence. San Mateo: Morgan Kaufmann.

Huhns, Michael N., and Les Gasser (eds.)
1989 Distributed Artificial Intelligence. San Mateo, CA: Morgan Kaufmann.

Kidder, Louise H.
1981 Research Methods in Social Relations (4th ed.). New York: Holt, Rinehart and Winston.

Lientz, Bennet P., and E. Burton Swanson
1980 Software Maintenance Management: A Study of the Maintenance of Computer Applications Software in 487 Data Processing Organizations. Reading, MA: Addison-Wesley.

Malone, Thomas W., and Kevin Crowston
1990 "What is coordination theory and how can it help design cooperative work systems?" In D. Tatar (ed.), Proceeding of the Third Conference on Computer Supported Cooperative Work. Los Angeles, CA: ACM Press.

Malone, Thomas W., Kenneth R. Grant, Franklyn A. Turbak, Steven A. Brobst, and Michael D. Cohen
1987a "Intelligent information-sharing systems." Communications of the ACM, 30: 390–402.

Malone, Thomas W., Joanne Yates, and Robert I. Benjamin
1987b "Electronic markets and electronic hierarchies." Communications of the ACM, 30: 484–497.

Marca, David A., and Clement L. McGowan
1988 SADT™: Structured Analysis and Design Technique. New York: McGraw-Hill.

March, James G., and Herbert A. Simon
1958 Organizations. New York, NY: John Wiley and Sons.

McDermott, Drew
1978 "Tarskian semantics, or No notation without denotation!" Cognive Science, 2: 277–282.

Moore, Robert C.
1979 "Reasoning About Knowledge and Action." Unpublished Ph.D. thesis. Department of Electric Engineering and Computer Science, Massachusetts Institute of Technology.

Moore, Robert C.
1982 "The role of logic in knowledge representation and commonsense reasoning." In Proceedings of AAAI National Conference on AI: 428–433. Pittsburgh, PA: AAAI.

Morgenstern, Leora
1987 "Knowledge preconditions for actions and plans." In Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87): 867–874

Morgenstern, Leora
1988 Foundations of a Logic of Knowledge, Action and Communication. Unpublished Ph. D. thesis, Department of Computer Science, New York University.

Newell, Allen, and Herbert. A. Simon
1972 Human Problem Solving. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Prietula, Michael J., Renee A. Beauclair, and F. Javier Lerch
1990 "A computation view of group problem solving." In Proceedings of the 23rd Hawaii International Conference on System Sciences. Kailua-Kona, Hawaii: IEEE Computer Society Press.

Ross, S.
1973 "The economic theory of agency." American Economic Review, 63: 134–139.

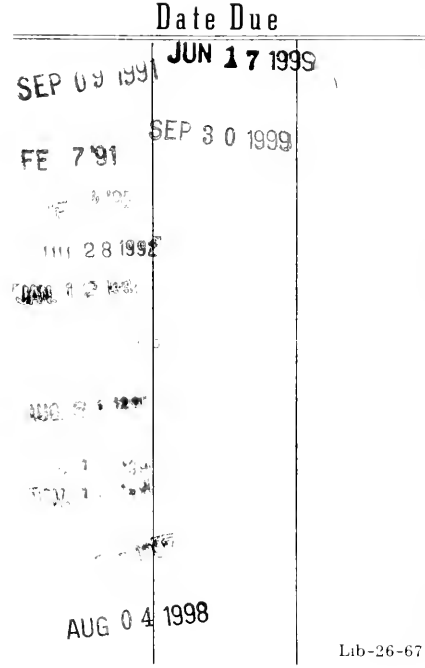
Star, Susan Leigh
1989 "The structure of ill-structured solutions: Boundary objects and heterogeneous distributed problem solving." In L. Gasser, and M. N. Huhns (eds.), Distributed Artificial Intelligence: 37–54. San Mateo, CA: Morgan Kaufmann.

Stefik, Mark, and Daniel G. Bobrow
1986 "Object-oriented programming: Themes and variations." AI Magazine: 40–62.

Thompson, James D.
1967 Organizations in Action: Social Science Bases of Administrative Theory. New York: McGraw-Hill.

Tushman, Michael, and David Nadler
1978 "Information processing as an integrating concept in organization design." Academy of Management Review, 3: 613–624.

Yourdon, Edward
1989 Modern Structured Analysis. Englewood Cliffs, NJ: Yourdon.


Date Due
SEP 09 1991
JUN 17 1998
SEP 30 1999
FE 7 91
JAN 28 1992
AUG 04 1998
Lib-26-67